Chapter 8. Sorting

8.0 Introduction

This chapter almost doesn't belong in a book on *numerical* methods. However, some practical knowledge of techniques for sorting is an indispensable part of any good programmer's expertise. We would not want you to consider yourself expert in numerical techniques while remaining ignorant of so basic a subject.

In conjunction with numerical work, sorting is frequently necessary when data (either experimental or numerically generated) are being handled. One has tables or lists of numbers, representing one or more independent (or "control") variables, and one or more dependent (or "measured") variables. One may wish to arrange these data, in various circumstances, in order by one or another of these variables. Alternatively, one may simply wish to identify the "median" value, or the "upper quartile" value of one of the lists of values. This task, closely related to sorting, is called *selection*.

Here, more specifically, are the tasks that this chapter will deal with:

- Sort, i.e., rearrange, an array of numbers into numerical order.
- Rearrange an array into numerical order while performing the corresponding rearrangement of one or more additional arrays, so that the correspondence between elements in all arrays is maintained.
- Given an array, prepare an *index table* for it, i.e., a table of pointers telling which number array element comes first in numerical order, which second, and so on.
- Given an array, prepare a *rank table* for it, i.e., a table telling what is the numerical rank of the first array element, the second array element, and so on.
- Select the Mth largest element from an array.

For the basic task of sorting N elements, the best algorithms require on the order of several times $N \log_2 N$ operations. The algorithm inventor tries to reduce the constant in front of this estimate to as small a value as possible. Two of the best algorithms are *Quicksort* (§8.2), invented by the inimitable C.A.R. Hoare, and *Heapsort* (§8.3), invented by J.W.J. Williams.

For large N (say > 1000), Quicksort is faster, on most machines, by a factor of 1.5 or 2; it requires a bit of extra memory, however, and is a moderately complicated program. Heapsort is a true "sort in place," and is somewhat more compact to program and therefore a bit easier to modify for special purposes. On balance, we recommend Quicksort because of its speed, but we implement both routines.

For small N one does better to use an algorithm whose operation count goes as a higher, i.e., poorer, power of N, if the constant in front is small enough. For $N < 20$, roughly, the method of *straight insertion* (§8.1) is concise and fast enough. We include it with some trepidation: It is an N^2 algorithm, whose potential for misuse (by using it for too large an N) is great. The resultant waste of computer time is so awesome, that we were tempted not to include any N^2 routine at all. We *will* draw the line, however, at the inefficient N^2 algorithm, beloved of elementary computer science texts, called *bubble sort*. If you know what bubble sort is, wipe it from your mind; if you don't know, make a point of never finding out!

For $N < 50$, roughly, *Shell's method* (§8.1), only slightly more complicated to program than straight insertion, is competitive with the more complicated Quicksort on many machines. This method goes as $N^{3/2}$ in the worst case, but is usually faster.

See references [1,2] for further information on the subject of sorting, and for detailed references to the literature.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1973, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley). [1]

Sedgewick, R. 1988, *Algorithms*, 2nd ed. (Reading, MA: Addison-Wesley), Chapters 8–13. [2]

8.1 Straight Insertion and Shell's Method

Straight insertion is an N^2 routine, and should be used only for small N, say < 20.

The technique is exactly the one used by experienced card players to sort their cards: Pick out the second card and put it in order with respect to the first; then pick out the third card and insert it into the sequence among the first two; and so on until the last card has been picked out and inserted.

```
      SUBROUTINE piksrt(n,arr)
      INTEGER n
      REAL arr(n)
        Sorts an array arr(1:n) into ascending numerical order, by straight insertion. n is input;
        arr is replaced on output by its sorted rearrangement.
      INTEGER i,j
      REAL a
      do 12 j=2,n                    Pick out each element in turn.
        a=arr(j)
        do 11 i=j-1,1,-1             Look for the place to insert it.
          if(arr(i).le.a)goto 10
          arr(i+1)=arr(i)
        enddo 11
        i=0
10      arr(i+1)=a                   Insert it.
      enddo 12
      return
      END
```

What if you also want to rearrange an array `brr` at the same time as you sort `arr`? Simply move an element of `brr` whenever you move an element of `arr`: